Impact of Event Logger on Causal Message Logging Protocols for Fault Tolerant MPI

Lemarinier Pierre, Bouteiller Aurelien, Herault Thomas, Krawezik Geraud, Cappello Franck

Impact of Event Logger on Causal Message Logging Protocols for Fault Tolerant MPI

Aurelien Bouteiller, Boris Collin, Thomas Herault, Pierre Lemarinier,
Franck Cappello
INRIA/LRI, Université Paris-Sud, Orsay, France
E-mail: { bouteill, collin, herault, lemarini, fci }@lri.fr
phone: (+33) 1 69 15 4222
fax: (+33) 1 69 15 4213

***Abstract*— Fault tolerance in MPI becomes a main issue in the HPC community. Several approaches are envisioned from user or programmer controlled fault tolerance to fully automatic fault detection and handling. For this last approach, several protocols have been proposed in the literature. In a recent paper, we have demonstrated that uncoordinated checkpointing tolerates higher fault frequency than coordinated checkpointing. Moreover causal message logging protocols have been proved the most efficient message logging technique. These protocols consist in piggybacking non deterministic events to computation message. Several protocols have been proposed in the literature. Their merits are usually evaluated from four metrics: a) piggybacking computation cost, b) piggyback size, c) applications performance and d) fault recovery performance. In this paper, we investigate the benefit of using a stable storage for logging message events in causal message logging protocols. To evaluate the advantage of this technique we implemented three protocols: 1) a classical causal message protocol proposed in Manetho, 2) a state of the art protocol known as LogOn, 3) a light computation cost protocol called Vcausal. We demonstrate a major impact of this stable storage for the three protocols, on the four criteria for micro benchmarks as well as for the NAS benchmark.**

I. Introduction

Recently, as the number of processors is increasing in parallel systems, the need for fault tolerant MPI implementations has been reactivated. Several research projects are investigating fault tolerance at different levels: network [1], system [2], applications [3]. Different strategies have been proposed to implement fault tolerance in MPI: a) user/programmer detection and management, b) pseudo automatic, guided by the programmer and c) fully automatic/transparent. For the last category, several protocols have been discussed in the literature. As a consequence, for the user and infrastructure administrator, there is a choice not only among a variety of fault tolerance approaches but also among various fault tolerance protocols.

We have demonstrated in precedent papers [4], [5] that among the different kind of automatic and transparent fault tolerant protocol, message logging tolerates higher fault frequency than coordinated checkpointing. The figure 1 recalls this result, presenting the increase of execution time for the NAS BT Benchmark on a 25 nodes cluster, according to an increasing fault frequency. When the execution time with faults relatively to the execution time without fault reaches a vertical slope, the application does not progress anymore.

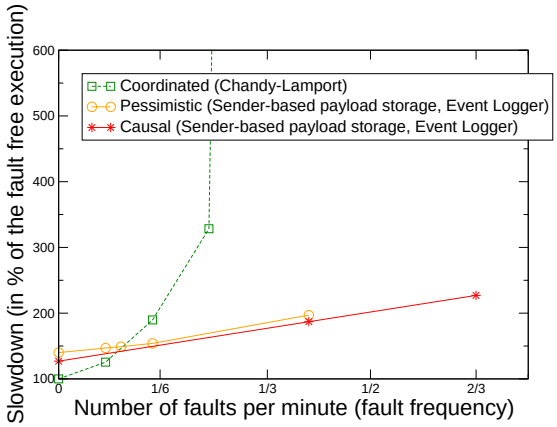


Fig. 1. Fault resilience comparison between Coordinated checkpoint, pessimistic message logging, and causal message logging protocols.

Among message logging protocols, we have demonstrated in [5] that causal message logging protocols are adapted to more applications and communication schemes.

In some previous works [6] [7], the use of a stable storage to log message events was suggested in order to reduce the memory occupation of protocol related information in computational nodes. In this paper we demonstrate the advantage of such stable storage called the Event Logger (EL) in the perspective of improving the performance of causal protocols. The EL helps to solve a major concern of causal message logging protocols which is the size of the causality information piggybacked in all messages. To demonstrate the advantages of using an EL, we implemented 3 causal message logging protocols. The first causal protocol is the reference protocol of the literature called Manetho. It relies on ordering an event structure to reduce piggyback size. The second protocol has only been discussed theoretically and is considered as an optimal one for the piggyback information construction. It consists in using a structure called LogOn to improve the piggyback management computation cost. The last protocol, called Vcausal, is a new one offering light piggyback computation cost at the expense of a weak piggyback size reduction. This last protocol is used to highlight the impact of the EL compared to strong piggyback reduction protocols. We demonstrate that the EL improves not only the performance of this naive protocol but also and significantly the performance of two other protocols (Manetho and LogOn) on the four

criteria usually considered to evaluate causal message logging protocols: time to manage piggyback information, size of piggybacked information, fault free applications performance and fault recovery performance. This result obtained for micro benchmarks as well as for the NAS benchmark suggests that EL is a fundamental component of causal message logging protocols.

The paper is organized as follows. The second part of the paper presents the related works highlighting the originality of this work. Section III presents the common principle of causal message logging protocols, the principle of the Event Logger (EL) and the piggyback management in the three considered protocols. Section IV presents the generic framework used to compare fault tolerant MPI protocols in a fair way and the implementation details of the three protocols. Section V presents the benefit of the EL, for the three protocols in terms of application performance and fault tolerance using micro benchmarks and the NAS benchmarks. Section VI sums up what we learned from these experiments.

II. Related work

Automatic and transparent fault tolerance techniques for message passing distributed systems have been studied for a long time. We can distinguish few classes of such protocols: replication protocols, rollback recovery protocols and self-stabilizing protocols. In replication techniques, every process is replicated f times. As a consequence, the system can tolerate less than f concurrent faults but divides the total computation resources by a factor of f. Self stabilizing techniques are used for non terminating computations. Rollback recovery protocols consist in taking checkpoint images of processes during initial execution and rolling back some processes to their last images when a failure occurs. These protocols take special care to respect the consistency of the execution in different manners. Rollback recovery protocols are the most studied techniques in the field of fault tolerant MPI. Several projects are working on implementing a fault tolerant MPI using different strategies and an overview can be found in [8].

Rollback recovery protocols include message logging protocols and global checkpoint techniques. Extended descriptions of these techniques can be found in [9].

Message logging protocols

Message logging consists in forcing the reexecution of crashed processes from their last checkpoint images to reach the state immediately preceding the crashing state, in order to recover a state coherent with non crashed ones. All message logging protocols suppose that the execution is *piecewise deterministic*. This means that the execution of a process in a distributed system is a sequence of deterministic and non deterministic events and is led by its non deterministic events. Most protocols suppose that the reception events are the only possible non deterministic events in an execution. Thus message logging protocols consist in logging all reception events of a process and in replaying the same sequence of receptions when this process crashes.

Three classes of message logging protocols are known: pessimistic, optimistic and causal message logging. Pessimistic message logging protocols ensure that all events of a process P are safely logged on stable storage before P can impact the system (sending a message) at the cost of synchronous operations. Optimistic protocols assume faults will not occur between an event and its logging, avoiding the need of synchronous operations. As a consequence, when a fault occurs, some non crashed processes may have to rollback. Causal protocols try to combine the advantages of both optimistic and pessimistic protocols: low performance overhead during failure free execution and no rollback of any non crashed process. This is realized by piggybacking events to messages until these events are safely logged. A formal definition of the three logging techniques may be found in [10]. Every causal protocol may use a reliable asynchronous remote storage intended to garbage collect causality events. However, although the potential use of this component is pointed out in several previous works, no evaluation of its impact on performance exists. The classical measures for complexity in causal protocols is the piggyback size and the time consumed to compute the causal information. In order to evaluate the impact of the reliable storage (called the Event Logger -EL- in this paper) by itself, we will compare the performance of different piggybacking reduction techniques, with or without this storage.

Piggybacking reduction techniques: We study the impact of the EL for three different reduction techniques. The first one is the one used in the Manetho project [6]. For the second one, we made the first implementation of the protocol presented in [7], this protocol is claimed to be optimal in term of number of events piggybacked, and improves time to build piggyback. The third one is our original method based on the asynchronous remote storage of causal event information presented in [5] and called Vcausal.

Manetho [6], [11] presents the first implementation of a causal message logging protocol. Each process maintains an *antecedence graph* which records the causal relationship between non deterministic events. In order to remove excess piggybacking, when a process sends a message to another one, it does not send the complete graph, but an incremental piggybacking: all events preceding one initially created by the receiver do not need to be sent back to it. This adds a computational overhead to latency.

The second considered protocol has been proposed in [7] to reduce the amount of information piggybacked on each message. It is referenced as LogOn in this paper. Like Manetho it is based on an antecedence graph. Additionally, it partially reorders events from a log inheritance relationship. This requires no additional piggybacking information and allows having some information about the causality a receiver may already hold.

An estimation of the overhead introduced by causal message logging protocols has been studied by simulation in [12]. This evaluation focuses on a comparison of various piggybacking elimination protocols, intended to reduce the amount of data

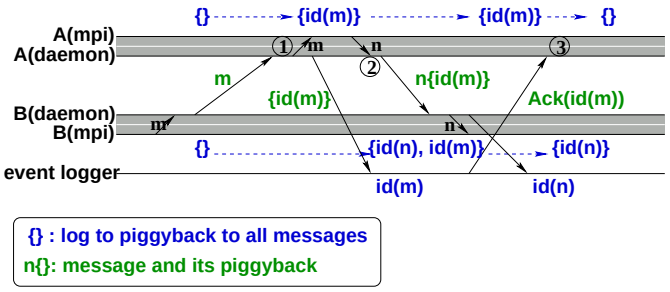


Fig. 2. Sample of execution without fault of a causal logging protocol with an Event Logger

piggybacked at the expense of considering a bound on the number of simultaneous failures. When a causal event has been replicated enough to ensure fault tolerance, it is no more piggybacked. All these protocols introduce some overhead in comparison to Manetho when considering that all processes may fail simultaneously.

A comparison between various fault tolerant protocols, including comparison between causal message logging protocols, pessimistic message logging protocols and coordinated checkpoint for MPI has been studied in [4], [5], using the MPICH-V framework. The MPICH-V framework is detailed in [13], [2]. Vcausal is an implementation of a causal logging protocol for MPICH-V. It uses a light computation cost piggybacking reduction technique, and relies on an EL to reduce the amount of piggybacked events.

In this previous work we did not investigate the impact of the EL on the application and fault tolerance performance. The specificities introduced in this framework to perform comparisons between various causal log piggybacking reduction techniques are presented in section IV.

Comparison between pessimistic and causal message logging has been studied using Egida [14] but there is no result about the benefit of using an Event Logger on the application and fault tolerance performance.

III. Causal message logging protocols

Every considered causal message logging protocol relies on a sender based approach. When a process sends a message, it stores its payload on its volatile memory. When a process is restarted, it requests all other processes to send back every message needed for its reexecution, with respect to the order of non deterministic events to replay. The three causal protocol implementations presented here use the same sender based causal message logging mechanism. The three protocols do not suppose any bound on the number of simultaneous faults in the system. Compared to pessimistic message logging, the basic principle of causal protocols is to relax the wait of acknowledgment from the stable repository by piggybacking events to computation messages. In causal protocols, a stable storage is not mandatory for fault tolerance. Instead, in previous works, it was only proposed to ensure garbage collection of events. The main criteria differentiating the three protocols studied in this paper is the technique used to reduce the size of piggybacked events.

A. Causal message logging with Event Logger

The three protocols act as the following (figure 2) with an Event Logger: when A receives a message from B ①, the node A associates a unique identifier to the reception (an event) and sends asynchronously this causality information to the Event Logger. When A has to send a message ②, the causality information of all previous receptions is added to the message only if they have not been acknowledged by the Event Logger yet. When the Event Logger acknowledges some causality information ③, this information is discarded in A. If A fails, it is restarted in its last checkpoint state. It collects from the EL and from every other alive nodes all the causality information and conforms its execution to this information until it reaches the same state as preceding the crash. Then, the execution continues normally.

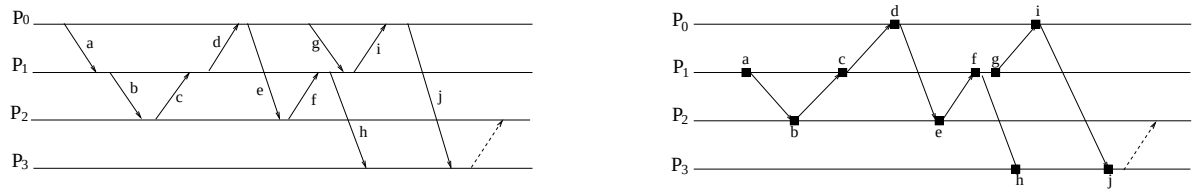
When the EL acknowledges the clock of stable events, all protocols assume they can garbage collect some event information. Note that the Manetho and LogOn antecedence graphs lose some vertices and incident edges, information avoiding the emission of unnecessary events. A contribution of this paper is to highlight the real impact of the use of an EL on performance of these various piggybacking management techniques.

We will demonstrate that the Event Logger will improve the performance of causal message logging protocol but at the cost of adding a stable component. However every fault tolerant protocol assume such a stable component at least to run a checkpoint server. The Event Logger can be run on the same node if the number of stable components in a system is restricted to 1. For a large application, with a high rate of communications, this would lead to decrease the performance of the Event Logger down to be useless. However, even for only checkpointing performance concerns, when the overall number of nodes is high, the bandwidth of a single reliable node may not be sufficient and implies using more than one reliable node. Moreover, the relative cost of using one supplementary dedicated node to fault tolerance is reduced when the number of computing nodes is high.

B. Event piggybacking management

1) Vcausal piggybacking reduction method: In the Vcausal protocol, each node uses one sequence of events per process to store the causality information. When a node A receives some causality information from a process B, it appends this information to its logs. Moreover it stores knowledge of the last events e_p, created by each process p, it has received from B. When A sends a message to B, it piggybacks every event from e_p to the end of its sequences and changes e_p to the last events it sends to B. As there is a total order between the causal events generated by a single node p, all events are propagated and no event are sent twice between two nodes.

2) Antecedence based piggybacking reduction method: Whereas Vcausal maintains only a reception sequence sorted by reception clock for each process, both Manetho and LogOn maintain an *antecedence graph*. This graph extends the reception sequences structure of Vcausal with a relation between



(a) A possible execution

(b) The associated antecedence graph

Fig. 3. Example of causal protocol execution and the resulting antecedence graph

events of different processes. Two events e_{P_1} of process P_1 and e_{P_2} of process P_2 are linked if and only if e_{P_2} denotes a reception of a message m sent by P_1 and e_{P_1} is the last non deterministic events preceding the emission of m. Thus crossing this graph allows to better estimate the events already known by a process. As in Vcausal, the Manetho and LogOn protocols ensure that no process sends any event to P_i that they have already sent to it.

The difference between Manetho and LogOn appears in the way they build the antecedence graph. When a process sends a message to a peer P_r, Manetho first searches for the last events P_r knows. To find this bound, the graph is crossed from the last known reception of P_r. All events happened after this bound have to be send to P_r.

LogOn reorders events according to the algorithm presented in [7]. It explores the antecedence graph in a reverse order, starting from the last reception event of the sender P_s, until reaching events from the receiver. After a reception, the antecedence graph is updated by crossing only one time the graph and the cost to maintain the antecedence relationship may be lower than for Manetho. Conversely, as Manetho does not rely on this partial order, it is necessary to first add the new piggybacked events, before generating new edges of the graph.

The two strategies for computing the border of the antecedence graph to be piggybacked induce different behaviors respectively to communication pattern of applications. Thus, time to serialize events to piggyback may differ between LogOn and Manetho.

Let consider the execution example and the resulting antecedence graph presented figure 3. The information piggybacked to the last message figured using dotted arrow is different for the three protocols. In Vcausal protocol, as P_3 has never received, neither sent anything to P_2, it will send all events to P_2. In Manetho and LogOn, using the antecedence graph, P_3 can compute the events P_2 already knows. So events from a to e are not piggybacked while events from f to j are.

C. Causal protocols implementation details

In the implementation of Vcausal and Manetho protocols, in order to reduce the piggybacked information size, the reception events are factored by peer rank. These two implementations use the same piggyback format: a list of $\{rid, nb, sequenceofevents\}$ where rid is the receiver rank of the event, and nb the number of events. Thus the receiver rank is not added to each event. LogOn uses a partial order to accelerate the processing on a piggyback reception: let

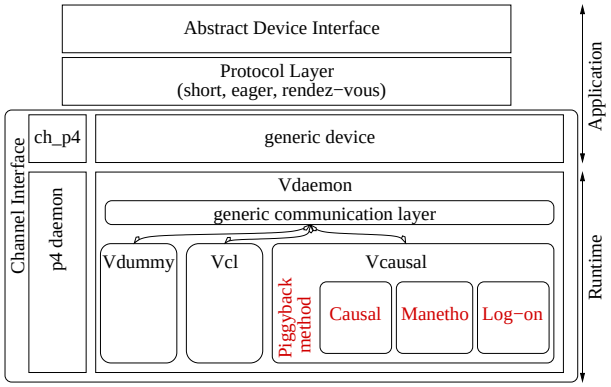


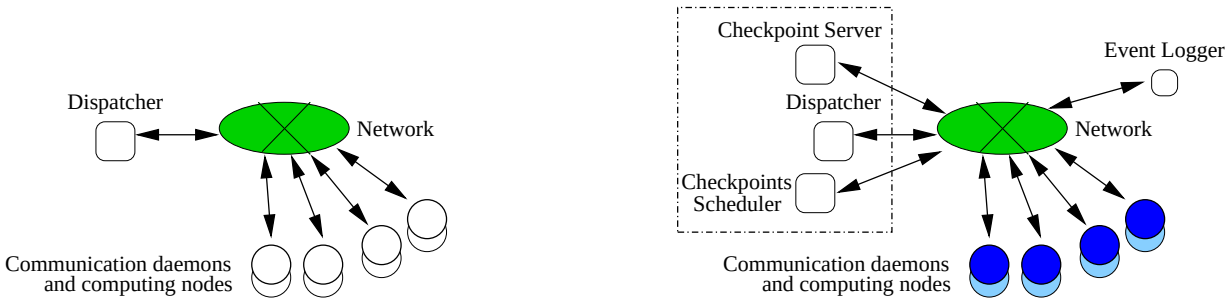
Fig. 4. General Architecture of MPICH-V compared to the architecture of MPICH-P4

$m_1, m_2, \ldots, m_k$ be a piggyback, for all $i, j; i < j$, m_j can not be in the past of m_i. Thus, in LogOn, it is not possible to factor events. As a consequence, each event of the piggyback sequence contains the receiver rank. As a consequence, for the same number of events to piggyback, the actual size in bytes of data added to the message is higher for LogOn.

IV. Generic fault tolerance framework

We designed a generic framework, named MPICH-V, to compare different fault tolerance protocols for MPI applications. MPICH-V is based on the MPICH 1.2.5 library [15], which builds a full MPI library from a channel. A channel implements the basic communication routines for a specific hardware or for new communication protocols. MPICH-V consists of a set of runtime components and a channel (ch_v) for the MPICH library.

The generic framework of MPICH-V relies on a separation of the MPI application and the actual communications on the network (figure 4). Communication daemons (Vdaemon) provide all the communication routines between the different kind of components involved in the MPICH-V architecture and are detailed below. Fault tolerance protocols are designed through the implementation of a set of hooks called in relevant routines of the generic subsystem and some specific components. We call *V-protocol* such an implementation. In this study in addition to causal message protocols (Vcausal) we consider an other V-protocol (Vdummy). Vdummy is a trivial implementation of these hooks which does not provide any fault tolerance (equivalent to the MPICH-P4 reference implementation). It is used to measure the raw performances of the generic communication layer. Figure 4 outlines where different causal techniques are implemented in the shared Vcausal V-protocol.



(a) Typical deployment of MPICH-Vdummy

(b) Typical deployments of MPICH-Vcausal (reliable components in white)

Fig. 5. Components of the MPICH-V framework

A. Communication daemon

In the generic architecture, the MPI process does not connect directly to the other ones. It communicates with a generic communication daemon, through a pair of system pipes. These daemons are connected together and relay the communications. This separation is mainly due to checkpoint-related constraints.

The daemon handles the effective communications, namely sending, receiving, reordering messages, establishing connections with all components of the system and detecting failures. In each of these routines, protocol dependent functions are called. The collection of all these functions is defined through a fault tolerance API and each fault tolerant protocol implements this API. In order to reduce the number of system calls, communications are packed using *iovec* related techniques by the generic communication layer. The different communication channels are multiplexed using a single thread and the *select* system call. This common implementation of communications allows a fair comparison between the different protocols.

B. Auxiliary stable servers

Four other components are involved in fault tolerance protocols: the dispatcher, the checkpoint server, the checkpoint scheduler and the Event Logger. Figures 5(a) and 5(b) present typical deployments of Vdummy and Vcausal runtime components. All these components should be run on a reliable system (potentially the same stable machine) and a more detailed description may be found in our previous papers on MPICH-V [13], [4], [5].

1) Dispatcher: The dispatcher of the MPICH-V environment has two main purposes: 1) to launch the whole runtime environment (encompassing the computing nodes and the auxiliary ”special” nodes) on the pool of machines used for the execution, and 2) to monitor this execution, by detecting any fault (node disconnection) and relaunching crashed MPI process instances possibly on available computing nodes.

2) Checkpoint server: The checkpoint server is a stable component storing the remote checkpoint images of each process. It is a multiprocess server, one process per client. All checkpoint operations (namely store, delete and retrieve of an image) are transactions: in case of a failure before the termination of the operation, the state of the checkpoint server and images is not modified. Note that in the case of message logging protocols, the checkpoint image of a process consists in the state of the MPI process, the payload of some messages and the causal information of all events stored in the local memory.

3) Checkpoint scheduler: The checkpoint scheduler is a specific component that is not necessary to insure the fault tolerance, but is intended to enhance performance. In message logging protocols, all checkpoints are taken uncoordinated and messages payload is stored in volatile memories of each sender process. When a checkpoint of a process is finished, the sender-based messages payload of all receptions preceding the checkpoint can be deleted. Thus, to increase the overall performance, it is important that checkpoint scheduling maximizes this garbage collecting. The checkpoint scheduler implements different policies such as coordinated checkpoint, random or round-robin.

4) Event Logger: The Event Logger is a component specific to the message logging protocols we developed. It acts as a reliable storage for all causality events of an execution. Every process sends asynchronously each reception event to the Event Logger. Then the Event Logger sends back an acknowledgment, notifying about the last event stored for each process. The Event Logger is a single thread server based on a *select* loop to handle non blocking asynchronous communications.

V. Experiments

A. Experimental conditions

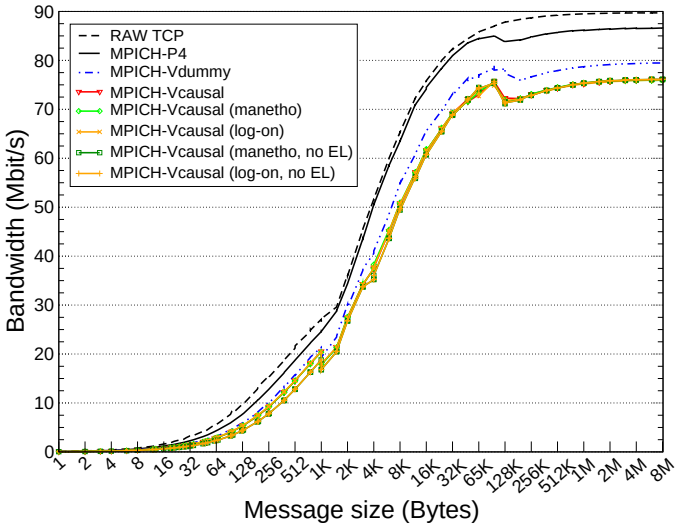
We present a set of experiments in order to evaluate the different components of the system.

Ethernet experiments are run on a 32-nodes cluster. Each node is equipped with an AthlonXP 2800+ processor, running at 2GHz, 1GB of main memory (DDR SDRAM), and a 70GB IDE ATA100 hard drive and a 100Mbit/s Ethernet Network Interface card. All nodes are connected by a single Fast Ethernet Switch.

Nodes use Linux 2.4.20 as operating system. The tests and benchmarks are compiled with GCC (with flag -O3) and the PGI Fortran77 compilers. All tests are run in dedicated mode. Each measurement is repeated 5 times and we present a mean of them. Where needed, 2 stable nodes are used: 1 hosts the checkpoint server and the other the Event Logger.

MPI implementation	P4	Vdummy	Vcausal *with EL*			Vcausal *no EL*		
Piggyback elimination	*Non fault tolerant*		Standard	Manetho	LogOn	Standard	Manetho	LogOn
Ethernet Latency (μs)	99.56	134.84	156.92	156.80	155.83	165.17	173.15	172.80

(a) Latency comparison over Ethernet 100Mbit/s



(b) Ping-pong bandwidth comparison over Ethernet 100Mbit/s

Fig. 6. Ping-pong bandwidth and latency comparison between Raw TCP, MPICH-P4, MPICH-Vdummy, MPICH-Vcausal with various causal log piggybacking reduction strategies, with and without Event Logger.

The first experiments are synthetic benchmarks analyzing the individual performance of the subcomponents. We use the NetPIPE [16] utility to measure bandwidth and latency. This is a ping-pong test for several message sizes and small perturbations around these sizes. The second set of experiments is the set of kernels and applications of the NAS Parallel Benchmark suite [17], written by the NASA NAS research center to test high performance parallel computers. These benchmarks cover a large panel of communication schemes: each benchmark tests a particular communication scheme, and communication computation ratio. CG benchmark presents heavy point-to-point latency driven communications; BT benchmark presents large point-to-point messages, and communications overlapped by computation; LU benchmark tests large number of large messages communications, FT benchmark presents all-to-all communication pattern.

To measure the piggyback statistics, we have included some probes in the implementation.

B. Overhead evaluation

First experiments are intended to validate performance and identify the overhead of the shared framework used to compare all fault tolerant protocols.

Figure 6 presents a comparison of the bandwidth and latency achieved by the Netpipe ping-pong test using raw TCP network between the reference implementation MPICH-P4, our non fault tolerant implementation Vdummy (outlining the framework overhead itself), and the three causal message logging protocols.

The comparison between MPICH-P4 and MPICH-Vdummy on Ethernet network outlines a slight bandwidth degradation and a 30% latency overhead. This is related to the current implementation of MPICH-V: the communication daemon is a separated process, inducing some memory copies and context switches. Using a daemon is fundamental to decouple the MPI execution from the network hardware features, in the perspective of MPI execution migration. We plan to address this overhead problem without losing the migration ability.

Nonetheless, these overheads are clearly identified. As we compare all fault tolerant protocols using the same shared framework, it is possible to distinguish which overheads are framework related, and which are protocol related. Moreover, we are able to measure the computing overhead for all protocol, as shown in figure 8. This overhead is only protocol dependent and not related to network.

C. Raw performance evaluation

Figure 6 also compares the three fault tolerant protocols using the Netpipe test, with or without Event Logger. In this ping-pong test, all the reduction protocols studied do not piggyback any causality information, excepted the last reception event. All other causality events are created previously by the other process itself and are not to be piggybacked.

Figure 6(b) presents the bandwidth of the three piggybacking reduction techniques. As in this ping-pong test, all protocols add the same amount of piggybacked causality to messages, the bandwidth is the same. The performance decrease compared to Vdummy is related to the sender-based message logging overhead.

The figure 6(a) presents a latency comparison between the three protocols, with or without Event Logger. When using an Event Logger, the latency of the three protocols is the same.

For standard Vcausal, using Event Logger, half of small messages do not carry any causal piggybacking (2397 of 4999

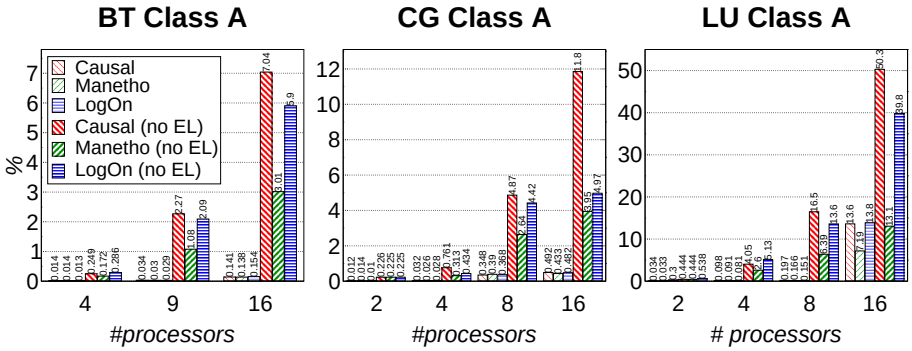


Fig. 7. Amount of piggybacked data exchanged during execution of BT, CG and LU class A NAS benchmarks in percent of the total exchanged data using Vcausal, Manetho and LogOn piggyback reduction techniques, with or without EL.

messages), while without Event Logger, every message carries one event, thus doubling the size of the message. The overall overhead on latency of not using an Event Logger for Vcausal is a 5.2% increase. The Event Logger decreases latency by reducing the amount of small messages carrying piggybacked events.

This overhead is 10.4% increase for antecedence graph based methods. When there is no Event Logger, additionally to the decreased number of empty messages, the size of the antecedence graph keeps growing on each node. In Vcausal, the reduction algorithm is very simple, while in antecedence graph based methods, the complete graph has to be traversed for each emission operation. The Event Logger decreases the latency by reducing the size of the antecedence graph in the node memory.

D. Fault free performance evaluation on NAS benchmarks

The size of the total information piggybacked is expected to be a main criteria to choose a causal message logging protocol as it has an impact on network performance. Another criteria is the computation time used to serialize and prepare the information piggybacked to computation message, and the time to add this information to the set of already known events when receiving such a message. The last fault free criteria is applications performance.

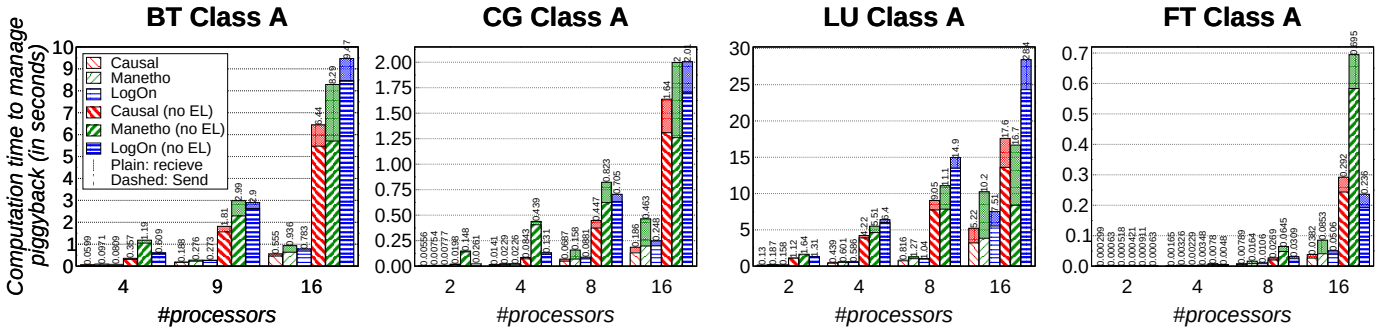
1) Comparison of the size of piggybacked data sent: Figure 7 presents the total amount of piggybacked information during execution of the NAS BT, CG and LU benchmarks for Vcausal, Manetho and LogOn protocols. When considering only performance without Event Logger, the LogOn protocol always piggybacks slightly more information except for LU benchmark for four nodes. As explained in section III-C, causality piggybacked in LogOn protocol cannot be factored, thus each event contains more information than in Manetho and Vcausal protocols. The LU benchmark for four nodes highlights the case where no factoring can be accomplished, as there is few information per piggyback. Vcausal protocol sends the highest amount of information due to its limited reduction technique. For the low communication/computation ratio of BT benchmark for 16 nodes, this amount represents 7% more data exchanged compared to a non fault tolerant implementation. It reaches 12% for CG for 16 nodes and even 51% for the high communication/computation ratio LU benchmark with 16 nodes. When an antecedence graph is used, this amount reaches 11% more data exchanged compared to non fault tolerant protocols, and only 7% when factoring events for the same LU benchmark.

When an EL is used, processes can erase causality information from their volatile memory as soon as it is acknowledged by this repository. For CG on 16 nodes, the Vcausal protocol sends only 4% of the total amount of piggybacked data sent when no Event Logger is used, and thus adds 0.5% more data than a non fault tolerant protocol. Manetho sends 14% of its volume without Event Logger and LogOn 6%. This outlines the major impact of using an Event Logger on the size of piggybacked events, even in the case of an optimized antecedence graph based protocol.

Figure 7 also presents the scalability of the three protocols, in term of amount of piggybacked data. For CG and BT benchmark, there is an exponential increase of the amount of data as the number of nodes grows. When using an Event Logger, this amount of information does not increase at the same rate.

In LU benchmark, a lot of messages are sent and the time between a reception and an emission is short. On LU benchmark when the number of nodes is less than 16, the size of piggybacked data does not increase with the number of nodes. With 16 nodes, the Event Logger reaches a state where the time to acknowledge event receptions becomes too high to remove all events before a new send occurs. However, for Vcausal, 70% of piggybacked data are eliminated, and for the antecedence graph based techniques, at least 50% of piggybacked data are saved. In our implementation, only one Event Logger is used for the whole application, leading to the stressing this process as the number of computation nodes increases.

2) Comparison of time to manage piggyback informations: Figure 8 presents the different times for the three protocols for CG, BT, LU and FT NAS benchmarks. Vcausal protocol does not require to cross any graph. It relies on an ordered sequence of events per neighbor. When preparing information to piggyback, all events of these sequences are added, from a set of particular events (see section III). Thus the Vcausal serialization outperforms the other two protocols. Manetho and LogOn protocols rely on crossing an antecedence graph to compute the minimum number of events to serialize. On



(a) Cumulative time to prepare causality information to piggyback when sending and time to add piggybacked information to known events when receiving

		With EL			Without EL		
		Vcausal	Manetho	LogOn	Vcausal	Manetho	LogOn
BT A	4	0%	0%	0%	0.2%	0.7%	0.2%
	9	0.2%	0.3%	0.3%	1.6%	3%	2.6%
	16	0.7%	1.3%	1.2%	7.8%	11.8%	12.5%
CG A	2	0%	0%	0.1%	0.2%	1.7%	0.3%
	4	0.1%	0.3%	0.3%	1%	5.1%	1%
	8	1%	2.5%	1.6%	6.8%	15%	11.2%
	16	2.4%	6.6%	4%	18%	26.1%	25.6%
LU A	2	0%	0%	0%	0.5%	0.7%	0.5%
	4	0.2%	0.4%	0.4%	2.9%	3.8%	3.8%
	8	0.9%	1.6%	1.4%	9.9%	12.2%	15%
	16	10.6%	19.1%	13.5%	26%	30.2%	41.5%
FT A	2	0%	0%	0%	0%	0%	0%
	4	0%	0%	0%	0%	0%	0%
	8	0%	0.1%	0%	0.1%	0.2%	0.1%
	16	0.3%	0.6%	0.4%	2.2%	5.2%	1.8%

(b) Causality information computation cost in percent of total execution time

Fig. 8. Time evaluation for extracting and serializing piggybacked causality information on BT, CG LU and FT class A NAS benchmarks using Vcausal, Manetho and LogOn piggyback reduction techniques, with or without EL.

all benchmarks, LogOn spends more time to reorder the resulting events for the serialization during send in order to accelerate the unserializing than Manetho. As a consequence, Manetho spends more time during receive to add new causality information to the antecedence graph. For CG and BT benchmarks, these two different strategies compare equally. In the case of LU the LogOn strategy is outperformed by Manetho due to the very large number of messages which decreases the performance of the serialization algorithm. The FT benchmark outlines that Manetho strategy is not suited for intensive global communications applications, while LogOn reaches good performance in this case.

Figure 8(b) presents the time spent to prepare causality information during send and the time to add causality data received, compared to the total execution time. This computation time appears to be an important aspect of the causal protocols for very high communication computation ratio applications. It reaches 41.5% of the total computation time for LogOn without Event Logger on LU class A on 16 nodes, and still 19.1% with Event Logger (using Manetho).

3) Applications performance: Figure 9 presents the performance of the NAS benchmarks on Ethernet, using MPICH-P4, MPICH-Vdummy and MPICH-Vcausal, with or without Event Logger, for the three causal protocols. Vdummy reaches better performance than P4 on some benchmarks. Unlike P4, Vdummy can benefit from full-duplex communications when the application communication scheme allows it.

When no Event Logger is used, the performance differences between causal protocols are larger. As expected, Vcausal performance without EL does not reach performance of antecedence graph based methods. Depending on the benchmark, Manetho or LogOn perform best. However, performance difference is small, at the exception of LU 16 where the large amount of piggybacked events decreases LogOn performance.

The impact of the EL is confirmed by this benchmark evaluation, especially for Vcausal protocol. The drastic diminution of number of piggybacked events induced by the introduction of the Event Logger increases bandwidth on all benchmarks. This leads Vcausal to compete with antecedence graph based methods when using Event Logger, at the exception of very high communication computation ratio applications. For antecedence graph based methods, in some cases (LU, CG), LogOn and Manetho performance ranking are inverted. LogOn benefits more than Manetho from the introduction of the EL, as on these benchmarks both size of exchanged piggybacked events and time to prepare piggyback is more reduced than for Manetho. Whatever the protocol or benchmark is used, performance is improved using Event Logger. The average performance improvement is greater than the average performance difference between the two antecedence graph based protocols.

E. Fault recovery performance

The three protocols share the same restart procedure. When an Event Logger is used, the events to replay can be retrieved from it. Without Event Logger, all events have to be reclaimed from all other computing nodes. Figure 10 presents time to recover events with or without Event Logger for Vcausal.

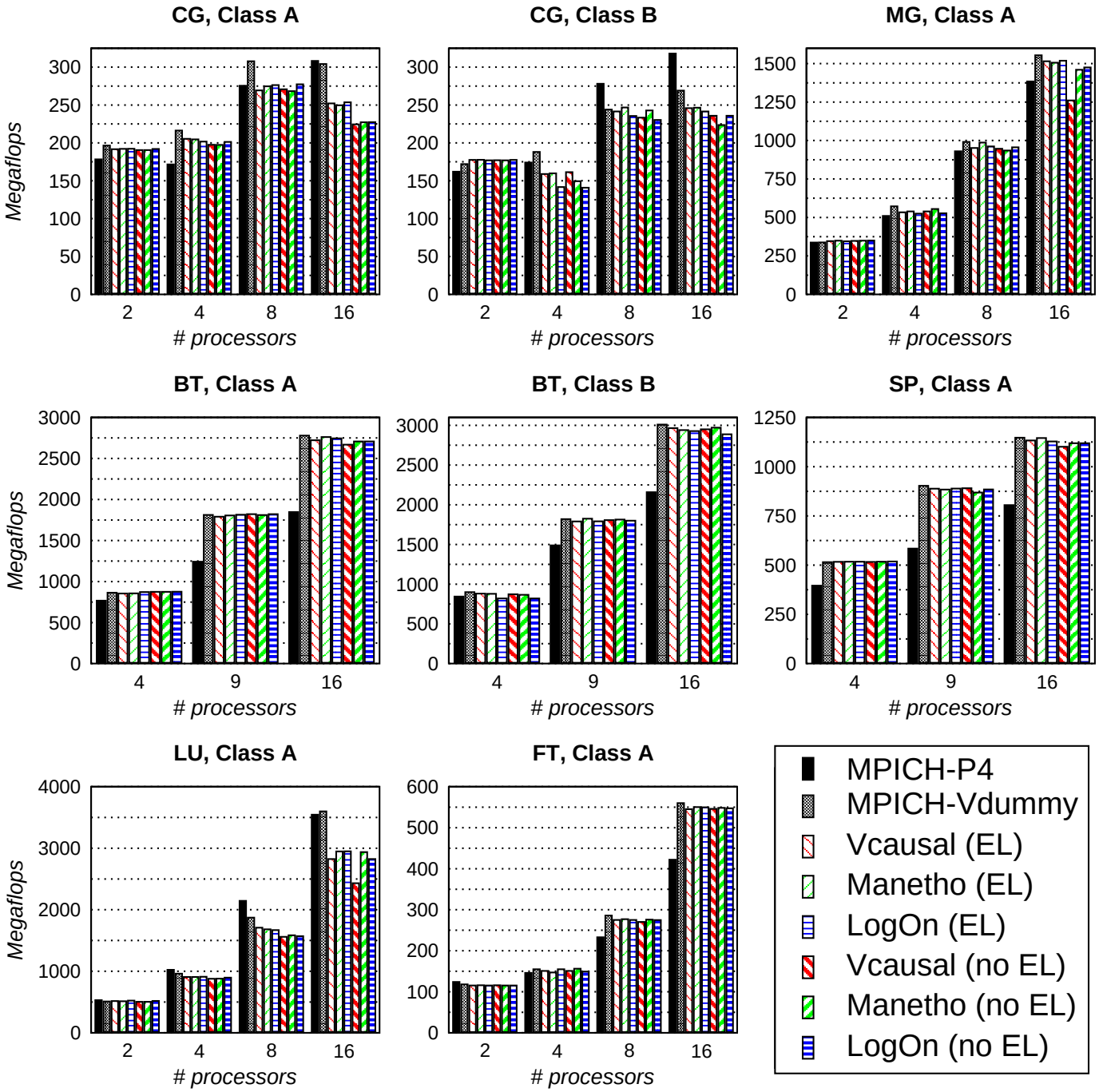


Fig. 9. Performance of the NAS benchmarks comparison using MPICH-P4, MPICH-Vdummy and MPICH-Vcausal, MPICH-Vcausal (Manetho), MPICH-Vcausal (LogOn) with or without Event Logger on Ethernet.

During the run of the benchmark, process of rank zero is killed at the middle of it's correct execution time and then restarted.

In BT, time to recover events using an Event Logger is 17.1% of time to retrieve them from other nodes for 9 processes, 11.8% for 16 processes, and 9.7% for 25 processes. CG and LU benchmarks outline the same behavior for these communication patterns.

On the CG benchmark, when using an Event Logger, time to recover causality events from 15 other nodes increases of 18.7% comparing to 1 other node. When not using an Event Logger, time to recover increases to 930.6%.

During restart, Event Logger improves performance. Additionally, the scalability is better addressed using Event Logger. This is due to the overall bandwidth consumption reduction implied by requesting events only once compared to reclaiming all events from all other nodes.

We recall that although the Event Logger should run on a stable node, it can be executed on the checkpoint server if

BT A	4	9	16	25
with EL	9.608	16.592	21.168	32.364
without EL	32.475	97.253	183.531	330.857
CG B	2	4	8	16
with EL	78.681	81.699	93.266	92.835
without EL	80.75	118.579	510.867	832.226
LU A	2	4	8	16
with EL	37.588	76.813	58.616	42.59
without EL	42.537	219.121	360.208	505.52

Fig. 10. Time (in milliseconds) to recover all events to replay when restarting using the Vcausal protocol on BT, CG, LU class A NAS benchmarks.

there is only one stable node in the system, at the expense of sharing the bandwidth.

VI. Conclusion and future works

Among automatic and transparent fault tolerance techniques on MPI, we proved in precedent papers [4], [5] that causal message logging protocols perform better than the other strategies for high fault frequencies. The purpose of this paper was to point out and study experimentally the impact of a stable component, the Event Logger, on the performance of causal message logging protocols. For this purpose, we compared three causal protocols for MPI applications with and without the presence of the Event Logger.

Using a generic framework we have implemented the three causal protocols: 1) Vcausal which relies on a simple set of sequences of reception events plus information about the last events sent or received from other nodes, 2) the Manetho protocol which uses an antecedence graph of events and traverses this graph to find the minimal amount of events other nodes have not logged yet, 3) the protocol described in [7] which relies on an antecedence graph and the reordering of piggybacks according to a partial order relationship to reduce the building time of the graph structure. We evaluated the impact of the Event Logger on these three protocols for Fast-Ethernet network. We demonstrated that the presence of the EL has a major impact on the four performance criteria: a) piggybacking computation cost, b) piggyback size, c) applications performance and d) fault recovery performance whatever is the causal message logging protocol considered. We also demonstrated that methods based on antecedence graph perform better than Vcausal basic techniques but use much more computation time handling causality.

Using only one Event Logger for consistency purpose will lead to a bottleneck as the number of processes grows. It is thus necessary to investigate how to distribute the logging of events among several Event Loggers. As all processes have to be notified of the reliable storage of an event in order to stop piggybacking this event, special care should be taken when distributing events among multiple Event Loggers. Assigning a subset of the nodes to one Event Logger seems the obvious way to gain scalability. But in order to keep the good performance introduced by the Event Logger in the system, each node has to receive the most up to date array of logical clocks already logged. In future work, we will compare different ways to design this distribution: by multicasting the local array of logical clocks of every Event Logger to the other ones, periodically or on specific events, and by broadcasting the local array of logical clocks of each Event Logger to all nodes and other Event Loggers. At last, this array of logical clocks may be piggybacked by the nodes, which will then communicate with a single Event Logger.

References

[1] S. Sankaran, J. M. Squyres, B. Barrett, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman, "The LAM/MPI checkpoint/restart framework: System-initiated checkpointing," in *Proceedings, LACSI Symposium*, Sante Fe, New Mexico, USA, October 2003.

[2] A. Bouteiller, F. Cappello, T. Hérault, G. Krawezik, P. Lemarinier, and F. Magniette, "MPICH-V2: a fault tolerant MPI for volatile nodes based on pessimistic sender based message logging," in *High Performance Networking and Computing (SC2003)*, Phoenix USA. IEEE/ACM, November 2003.

[3] G. Fagg and J. Dongarra, "FT-MPI : Fault tolerant MPI, supporting dynamic applications in a dynamic world," in *7th Euro PVM/MPI User's Group Meeting2000*, vol. 1908 / 2000. Balatonfred, Hungary: Springer-Verlag Heidelberg, september 2000.

[4] A. Bouteiller, P. Lemarinier, G. Krawezik, and F. Cappello, "Coordinated checkpoint versus message log for fault tolerant MPI," in *IEEE International Conference on Cluster Computing (Cluster 2003)*. IEEE CS Press, December 2003, pp. 242–250.

[5] P. Lemarinier, A. Bouteiller, T. Herault, G. Krawezik, and F. Cappello, "Improved message logging versus improved coordinated checkpointing for fault tolerant MPI," in *IEEE International Conference on Cluster Computing (Cluster 2004)*. IEEE CS Press, September 2004.

[6] Elnozahy, Elmootazbellah, and Zwaenepoel, "Manetho: Transparent rollback-recovery with low overhead, limited rollback and fast output," *IEEE Transactions on Computers*, vol. 41, no. 5, May 1992.

[7] B. Lee, T. Park, H. Y. Yeom, and Y. Cho, "An efficient algorithm for causal message logging," in *17th Symposium on Reliable Distributed Systems (SRDS 1998)*. IEEE CS Press, October 1998, pp. 19–25.

[8] W. Gropp and E. Lusk, "Fault tolerance in MPI programs," *special issue of the Journal High Performance Computing Applications (IJHPCA)*, 2002.

[9] M. Elnozahy, L. Alvisi, Y. M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Computing Surveys (CSUR)*, vol. 34, no. 3, pp. 375 – 408, september 2002.

[10] L. Alvisi and K. Marzullo, "Message logging : Pessimistic, optimistic, and causal," in *Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS 1995)*. IEEE CS Press, May-June 1995, pp. 229–236.

[11] E. N. Elnozahy and W. Zwaenepoel, "Replicated distributed processes in manetho," in *22nd International Symposium on Fault Tolerant Computing (FTCS-22)*. Boston, Massachusetts: IEEE Computer Society Press, 1992, pp. 18–27.

[12] K. Bhatia, K. Marzullo, and L. Alvisi, "The relative overhead of piggybacking in causal message logging protocols," in *17th Symposium on Reliable Distributed Systems (SRDS'98)*. IEEE CS Press, 1998, pp. 348–353.

[13] G. Bosilca, A. Bouteiller, F. Cappello, S. Djilali, G. Fédak, C. Germain, T. Hérault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Néri, and A. Selikhov, "MPICH-V: Toward a scalable fault tolerant MPI for volatile nodes," in *High Performance Networking and Computing (SC2002)*. Baltimore USA: IEEE/ACM, November 2002.

[14] S. Rao, L. Alvisi, and H. M. Vin, "The cost of recovery in message logging protocols," in *17th Symposium on Reliable Distributed Systems (SRDS)*. IEEE CS Press, October 1998, pp. 10–18.

[15] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "High-performance, portable implementation of the MPI message passing interface standard," *Parallel Computing*, vol. 22, no. 6, pp. 789–828, September 1996.

[16] Q. Snell, A. Mikler, and J. Gustafson, "Netpipe: A network protocol independent performance evaluator," in *IASTED International Conference on Intelligent Information Management and Systems.*, June 1996.

[17] D. Bailey, T. Harris, W. Saphir, R. V. D. Wijngaart, A. Woo, and M. Yarrow, "The NAS Parallel Benchmarks 2.0," Numerical Aerodynamic Simulation Facility, NASA Ames Research Center, Report NAS-95-020, 1995.